Microsoft BASIC is the most extensive implementation of BASIC available for microprocessors. Microsoft BASIC meets the ANSI qualifications for BASIC, as set forth in document BSRX3.60-1978. Each release of Microsoft BASIC is compatible with previous versions.

This manual is a reference for all implementations of Microsoft BASIC and for the Microsoft BASIC Compilers.

The manual is divided into three large chapters plus three appendices. Chapter 1 covers a variety of topics, largely pertaining to data representation in Microsoft BASIC. Chapter 2 contains the syntax and semantics of every command and statement in Microsoft BASIC, ordered alphabetically. Chapter 3 describes all of Microsoft BASIC's intrinsic functions, also ordered alphabetically. The appendices contain a list of error messages and codes, a list of mathematical functions, and a list of ASCII character codes.

Other useful information about programming Microsoft BASIC is covered in the Microsoft BASIC User's Guide, which describes which features of Microsoft BASIC are implemented for your machine, plus information relevant to your operating system, and helpful hints about such matters as data I/O and assembly language subroutines.

CHAPTER 1

GENERAL INFORMATION ABOUT MICROSOFT BASIC

1.1 INITIALIZATION

The procedure for initialization will vary with different implementations of Microsoft BASIC. Check the Microsoft BASIC User's Guide for your machine to determine how Microsoft BASIC is initialized with your operating system.

1.2 MODES OF OPERATION

When Microsoft BASIC is initialized, it types the prompt "Ok". "Ok" means BASIC is at command level; that is, it is ready to accept commands. At this point, Microsoft BASIC may be used in either of two modes: the direct mode or the indirect mode.

In the direct mode, BASIC statements and commands are not preceded by line numbers. They are executed as they are entered. Results of arithmetic and logical operations may be displayed immediately and stored for later use, but the instructions themselves are lost after execution. This mode is useful for debugging and for using BASIC as a "calculator" for quick computations that do not require a complete program.

The indirect mode is the mode used for entering programs. Program lines are preceded by line numbers and are stored in memory. The program stored in memory is executed by entering the RUN command.

1.3 LINE FORMAT

Program lines in a BASIC program have the following format (square brackets indicate optional):

nnnnn BASIC statement[:BASIC statement...] <carriage return>

At the programmer's option, more than one BASIC statement may be placed on a line, but each statement on a line must be separated from the last by a colon.

A Microsoft BASIC program line always begins with a line number, ends with a carriage return, and may contain a maximum of 255 characters.

It is possible to extend a logical line over more than one physical line by use of the terminal's <line feed> key. <Line feed> lets you continue typing a logical line on the next physical line without entering a <carriage return>.

1.3.1 Line Numbers

Every BASIC program line begins with a line number. Line numbers indicate the order in which the program lines are stored in memory and are also used as references when branching and editing. Line numbers must be in the range 0 to 65529. A period (.) may be used in EDIT, LIST, AUTO and DELETE commands to refer to the current line.

1.4 CHARACTER SET

The Microsoft BASIC character set is comprised of alphabetic characters, numeric characters and special characters.

The alphabetic characters in Microsoft BASIC are the upper case and lower case letters of the alphabet.

The numeric characters in Microsoft BASIC are the digits 0 through 9.

The following special characters and terminal keys are recognized by Microsoft BASIC:

Character	Name
	Blank
=	Equal sign or assignment symbol
+	Plus sign
-	Minus sign
*	Asterisk or multiplication symbol
/	Slash or division symbol
^	Up arrow or exponentiation symbol
(	Left parenthesis
)	Right parenthesis
%	Percent
#	Number (or pound) sign
$	Dollar sign
!	Exclamation point
[	Left bracket
]	Right bracket
,	Comma
.	Period or decimal point
'	Single quotation mark (apostrophe)
;	Semicolon
:	Colon
&	Ampersand
?	Question mark
<	Less than
>	Greater than
\	Backslash or integer division symbol
@	At-sign
_	Underscore
<rubout>	Deletes last character typed.
<escape>	Escapes Edit Mode subcommands. See Section 2.16.
<tab>	Moves print position to next tab stop. Tab stops are every eight columns.
<line feed>	Moves to next physical line.
<carriage return>	Terminates input of a line.

1.4.1 Control Characters

The following control characters are in Microsoft BASIC:

Control-A	Enters Edit Mode on the line being typed.
Control-C	Interrupts program execution and returns to BASIC command level.
Control-G	Rings the bell at the terminal.
Control-H	Backspace. Deletes the last character typed.
Control-I	Tab. Tab stops are every eight columns.
Control-O	Halts program output while execution continues. A second Control-O restarts output.
Control-R	Retypes the line that is currently being typed.
Control-S	Suspends program execution.
Control-Q	Resumes program execution after a Control-S.
Control-U	Deletes the line that is currently being typed.

1.5 CONSTANTS

Constants are the actual values BASIC uses during execution. There are two types of constants: string and numeric.

A string constant is a sequence of up to 255 alphanumeric characters enclosed in double quotation marks. Examples of string constants:

```
"HELLO"
"$25,000.00"
"Number of Employees"
```

Numeric constants are positive or negative numbers. Numeric constants in BASIC cannot contain commas. There are five types of numeric constants:

1.	Integer constants	Whole numbers between -32768 and +32767. Integer constants do not have decimal points.
2.	Fixed Point constants	Positive or negative real numbers, i.e., numbers that contain decimal points.

3. Floating Point constants

Positive or negative numbers represented in exponential form (similar to scientific notation). A floating point constant consists of an optionally signed integer or fixed point number (the mantissa) followed by the letter E and an optionally signed integer (the exponent). The allowable range for floating point constants is 10-38 to 10+38.
Examples:

235.988E-7 = .0000235988
2359E6 = 2359000000

(Double precision floating point constants use the letter D instead of E. See Section 1.5.1.)

4. Hex constants

Hexadecimal numbers with the prefix &H. Examples:

&H76
&H32F

5. Octal constants

Octal numbers with the prefix &O or &. Examples:

&O347
&1234

1.5.1 Single And Double Precision Form For Numeric Constants

Numeric constants may be either single precision or double precision numbers. Single precision numeric constants are stored with 7 digits of precision, and printed with up to 6 digits. With double precision, the numbers are stored with 16 digits of precision, and printed with up to 16 digits.

A single precision constant is any numeric constant that has:

1. seven or fewer digits, or
2. exponential form using E, or
3. a trailing exclamation point (!)

A double precision constant is any numeric constant that has:

1. eight or more digits, or
2. exponential form using D, or
3. a trailing number sign (#)

Examples:

Single Precision Constants	Double Precision Constants
46.8	345692811
-1.09E-06	-1.09432D-06
3489.0	3489.0#
22.5!	7654321.1234

1.6 VARIABLES

Variables are names used to represent values that are used in a BASIC program. The value of a variable may be assigned explicitly by the programmer, or it may be assigned as the result of calculations in the program. Before a variable is assigned a value, its value is assumed to be zero.

1.6.1 Variable Names And Declaration Characters

Microsoft BASIC variable names may be any length; up to 40 characters are significant. The characters allowed in a variable name are letters and numbers, and the decimal point. The first character must be a letter. Special type declaration characters are also allowed -- see below.

A variable name may not be a reserved word, but embedded reserved words are allowed. Embedded reserved words are allowed. If a variable begins with FN, it is assumed to be a call to a user-defined function. Reserved words include all Microsoft BASIC commands, statements, function names and operator names.

Variables may represent either a numeric value or a string.

String variable names are written with a dollar sign ($) as the last character. For example: A$ = "SALES REPORT". The dollar sign is a variable type declaration character, that is, it "declares" that the variable will represent a string.

Numeric variable names may declare integer, single or double precision values. The type declaration characters for these variable names are as follows:

%	Integer variable
!	Single precision variable
#	Double precision variable

The default type for a numeric variable name is single precision.

Examples of Microsoft BASIC variable names follow.

PI#	declares a double precision value
MINIMUM!	declares a single precision value
LIMIT%	declares an integer value
N$	declares a string value
ABC	represents a single precision value

There is a second method by which variable types may be declared. The BASIC statements DEFINT, DEFSTR, DEFSNG and DEFDBL may be included in a program to declare the types for certain variable names. These statements are described in detail in Section 2.12.

1.6.2 Array Variables

An array is a group or table of values referenced by the same variable name. Each element in an array is referenced by an array variable that is subscripted with an integer or an integer expression. An array variable name has as many subscripts as there are dimensions in the array. For example V(10) would reference a value in a one-dimension array, T(1,4) would reference a value in a two-dimension array, and so on. The maximum number of dimensions for an array is 255. The maximum number of elements per dimension is 32767.

1.6.3 Space Requirements

VARIABLES:	BYTES
INTEGER	2
SINGLE PRECISION	4
DOUBLE PRECISION	8

ARRAYS:	BYTES
INTEGER	2 per element
SINGLE PRECISION	4 per element
DOUBLE PRECISION	8 per element

STRINGS:

3 bytes overhead plus the present contents of the string.

1.7 TYPE CONVERSION

When necessary, BASIC will convert a numeric constant from one type to another. The following rules and examples should be kept in mind.

1. If a numeric constant of one type is set equal to a numeric variable of a different type, the number will be stored as the type declared in the variable name. (If a string variable is set equal to a numeric value or vice versa, a "Type mismatch" error occurs.)
   Example:

```
10 A% = 23.42
20 PRINT A%
RUN
 23
```

2. During expression evaluation, all of the operands in an arithmetic or relational operation are converted to the same degree of precision, i.e., that of the most precise operand. Also, the result of an arithmetic operation is returned to this degree of precision.
   Examples:

```
10 D# = 6#/7      The arithmetic was performed
20 PRINT D#       in double precision and the
RUN               result was returned in D#
 .8571428571428571 as a double precision value.
```

```
10 D = 6#/7
20 PRINT D
RUN
 .857143
```

The arithmetic was performed in double precision and the result was returned to D (single precision variable), rounded and printed as a single precision value.

3. Logical operators (see Section 1.8.3) convert their operands to integers and return an integer result. Operands must be in the range -32768 to 32767 or an "Overflow" error occurs.

4. When a floating point value is converted to an integer, the fractional portion is rounded. Example:

```
10 C% = 55.88
20 PRINT C%
RUN
 56
```

5. If a double precision variable is assigned a single precision value, only the first seven digits, rounded, of the converted number will be valid. This is because only seven digits of accuracy were supplied with the single precision value. The absolute value of the difference between the printed double precision number and the original single precision value will be less than 6.3E-8 times the original single precision value. Example:

```
10 A = 2.04
20 B# = A
30 PRINT A;B#
RUN
 2.04  2.039999961853027
```

1.8 EXPRESSIONS AND OPERATORS

An expression may be simply a string or numeric constant, or a variable, or it may combine constants and variables with operators to produce a single value.

Operators perform mathematical or logical operations on values. The operators provided by BASIC may be divided into four categories:

1. Arithmetic

2. Relational

3. Logical

4. Functional

1.8.1 Arithmetic Operators

The arithmetic operators, in order of precedence, are:

Operator	Operation	Sample Expression
^	Exponentiation	X^Y
-	Negation	-X
*,/	Multiplication, Floating Point Division	X*Y X/Y
+,-	Addition, Subtraction	X+Y

To change the order in which the operations are performed, use parentheses. Operations within parentheses are performed first. Inside parentheses, the usual order of operations is maintained.

Here are some sample algebraic expressions and their BASIC counterparts.

Algebraic Expression	BASIC Expression	
X+2Y	X+Y*2	
X-	X-Y/Z	
	X*Y/Z	
	(X+Y)/Z	
(X	(X^2)^Y	
X	X^(Y^Z)	
X(-Y)	X*(-Y)	Two consecutive operators must be separated by parentheses.

1.8.1.1 Integer Division And Modulus Arithmetic -
Two additional operators are available in Microsoft BASIC: Integer division and modulus arithmetic.

Integer division is denoted by the baskslash (\). The operands are rounded to integers (must be in the range -32768 to 32767) before the division is performed, and the quotient is truncated to an integer.
For example:

```
10\4 = 2
25.68\6.99 = 3
```

The precedence of integer division is just after multiplication and floating point division.

Modulus arithmetic is denoted by the operator MOD. It gives the integer value that is the remainder of an integer division. For example:

```
10.4 MOD 4 = 2 (10/4=2 with a remainder 2)
25.68 MOD 6.99 = 5 (26/7=3 with a remainder 5)
```

The precedence of modulus arithmetic is just after integer division.

1.8.1.2 Overflow And Division By Zero -
If, during the evaluation of an expression, a division by zero is encountered, the "Division by zero" error message is displayed, machine infinity with the sign of the numerator is supplied as the result of the division, and execution continues. If the evaluation of an exponentiation results in zero being raised to a negative power, the "Division by zero" error message is displayed, positive machine infinity is supplied as the result of the exponentiation, and execution continues.

If overflow occurs, the "Overflow" error message is displayed, machine infinity with the algebraically correct sign is supplied as the result, and execution continues.

1.8.2 Relational Operators

Relational operators are used to compare two values. The result of the comparison is either "true" (-1) or "false" (0). This result may then used to make a decision regarding program flow. (See IF, Section 2.26.)

Operator	Relation Tested	Expression
=	Equality	X=Y
<>	Inequality	X<>Y
<	Less than	X<Y
>	Greater than	X>Y
<=	Less than or equal to	X<=Y
>=	Greater than or equal to	X>=Y

(The equal sign is also used to assign a value to a variable. See LET, Section 2.30.)

When arithmetic and relational operators are combined in one expression, the arithmetic is always performed first. For example, the expression

```
X+Y < (T-1)/Z
```

is true if the value of X plus Y is less than the value of T-1 divided by Z. More examples:

```
IF SIN(X)<0 GOTO 1000
IF I MOD J <> 0 THEN K=K+1
```

1.8.3 Logical Operators

Logical operators perform tests on multiple relations, bit manipulation, or Boolean operations. The logical operator returns a bitwise result which is either "true" (not zero) or "false" (zero). In an expression, logical operations are performed after arithmetic and relational operations. The outcome of a logical operation is determined as shown in the following table. The operators are listed in order of precedence.

NOT

X	NOT X
1	0
0	1

AND

X	Y	X AND Y
1	1	1
1	0	0
0	1	0
0	0	0

OR

X	Y	X OR Y
1	1	1
1	0	1
0	1	1
0	0	0

XOR

X	Y	X XOR Y
1	1	0
1	0	1
0	1	1
0	0	0

EQV

X	Y	X EQV Y
1	1	1
1	0	0
0	1	0
0	0	1

IMP

X	Y	X IMP Y
1	1	1
1	0	0
0	1	1
0	0	1

Just as the relational operators can be used to make decisions regarding program flow, logical operators can connect two or more relations and return a true or false value to be used in a decision (see IF, Section 2.26). For example:

```
IF D<200 AND F<4 THEN 80
IF I>10 OR K<0 THEN 50
IF NOT P THEN 100
```

Logical operators work by converting their operands to sixteen bit, signed, two's complement integers in the range -32768 to +32767. (If the operands are not in this range, an error results.) If both operands are supplied as 0 or -1, logical operators return 0 or -1. The given operation is

performed on these integers in bitwise fashion, i.e., each bit of the result is determined by the corresponding bits in the two operands.

Thus, it is possible to use logical operators to test bytes for a particular bit pattern. For instance, the AND operator may be used to "mask" all but one of the bits of a status byte at a machine I/O port. The OR operator may be used to "merge" two bytes to create a particular binary value. The following examples will help demonstrate how the logical operators work.

63 AND 16=16	63 = binary 111111 and 16 = binary 10000, so 63 AND 16 = 16
15 AND 14=14	15 = binary 1111 and 14 = binary 1110, so 15 AND 14 = 14 (binary 1110)
-1 AND 8=8	-1 = binary 1111111111111111 and 8 = binary 1000, so -1 AND 8 = 8
4 OR 2=6	4 = binary 100 and 2 = binary 10, so 4 OR 2 = 6 (binary 110)
10 OR 10=10	10 = binary 1010, so 1010 OR 1010 = 1010 (10)
-1 OR -2=-1	-1 = binary 1111111111111111 and -2 = binary 1111111111111110, so -1 OR -2 = -1. The bit complement of sixteen zeros is sixteen ones, which is the two's complement representation of -1.
NOT X=-(X+1)	The two's complement of any integer is the bit complement plus one.

1.8.4 Functional Operators

A function is used in an expression to call a predetermined operation that is to be performed on an operand. Microsoft BASIC has "intrinsic" functions that reside in the system, such as SQR (square root) or SIN (sine). All of Microsoft BASIC's intrinsic functions are described in Chapter 3.

Microsoft BASIC also allows "user defined" functions that are written by the programmer. See DEF FN, Section 2.11.

1.8.5 String Operations

Strings may be concatenated using +. For example:

```
10 A$="FILE" : B$="NAME"
20 PRINT A$ + B$
30 PRINT "NEW " + A$ + B$
RUN
FILENAME
NEW FILENAME
```

Strings may be compared using the same relational operators that are used with numbers:

```
=    <>    <    >    <=    >=
```

String comparisons are made by taking one character at a time from each string and comparing the ASCII codes. If all the ASCII codes are the same, the strings are equal. If the ASCII codes differ, the lower code number precedes the higher. If, during string comparison, the end of one string is reached, the shorter string is said to be smaller. Leading and trailing blanks are significant. Examples:

```
"AA" < "AB"
"FILENAME" = "FILENAME"
"X&" > "X#"
"CL " > "CL"
"kg" > "KG"
"SMYTH" < "SMYTHE"
B$ < "9/12/78"      where B$ = "8/12/78"
```

Thus, string comparisons can be used to test string values or to alphabetize strings. All string constants used in comparison expressions must be enclosed in quotation marks.

1.9 INPUT EDITING

If an incorrect character is entered as a line is being typed, it can be deleted with the RUBOUT key or with Control-H. Rubout surrounds the deleted character(s) with backslashes, and Control-H has the effect of backspacing over a character and erasing it. Once a character(s) has been deleted, simply continue typing the line as desired.

To delete a line that is in the process of being typed, type Control-U. A carriage return is executed automatically after the line is deleted.

To correct program lines for a program that is currently in memory, simply retype the line using the same line number. BASIC will automatically replace the old line with the new line.

More sophisticated editing capabilities are provided. See EDIT, Section 2.16.

To delete the entire program that is currently residing in memory, enter the NEW command. (See Section 2.41.) NEW is usually used to clear memory prior to entering a new program.

1.10 ERROR MESSAGES

If BASIC detects an error that causes program execution to terminate, an error message is printed. For a complete list of Microsoft BASIC error codes and error messages, see Appendix A.

CHAPTER 2

MICROSOFT BASIC COMMANDS AND STATEMENTS

All of the Microsoft BASIC commands and statements are described in this chapter. Each description is formatted as follows:

Format: Shows the correct format for the instruction. See below for format notation.

Purpose: Tells what the instruction is used for.

Remarks: Describes in detail how the instruction is used.

Example: Shows sample programs or program segments that demonstrate the use of the instruction.

Format Notation

Wherever the format for a statement or command is given, the following rules apply:

1. Items in capital letters must be input as shown.

2. Items in lower case letters enclosed in angle brackets (< >) are to be supplied by the user.

3. Items in square brackets ([ ]) are optional.

4. All punctuation except angle brackets and square brackets (i.e., commas, parentheses, semicolons, hyphens, equal signs) must be included where shown.

5. Items followed by an ellipsis (...) may be repeated any number of times (up to the length of the line).

2.1 AUTO

Format: AUTO [<line number>[,<increment>]]

Purpose: To generate a line number automatically after every carriage return.

Remarks: AUTO begins numbering at <line number> and increments each subsequent line number by <increment>. The default for both values is 10. If <line number> is followed by a comma but <increment> is not specified, the last increment specified in an AUTO command is assumed.

If AUTO generates a line number that is already being used, an asterisk is printed after the number to warn the user that any input will replace the existing line. However, typing a carriage return immediately after the asterisk will save the line and generate the next line number.

AUTO is terminated by typing Control-C. The line in which Control-C is typed is not saved. After Control-C is typed, BASIC returns to command level.

Example:

AUTO 100,50	Generates line numbers 100, 150, 200 ...
AUTO	Generates line numbers 10, 20, 30, 40 ...

2.2 CALL

Format: CALL <variable name>[(<argument list>)]

Purpose: To call an assembly language subroutine.

Remarks: The CALL statement is one way to transfer program flow to an external subroutine. (See also the USR function, Section 3.40)

<variable name> contains an address that is the starting point in memory of the subroutine. <variable name> may not be an array variable name. <argument list> contains the arguments that are passed to the external subroutine. <argument list> may contain only variables.

The CALL statement generates the same calling sequence used by Microsoft's FORTRAN, COBOL and BASIC compilers.

Example:
```
110 MYROUT=&HD000
120 CALL MYROUT(I,J,K)
     .
     .
     .
```

NOTE: For a BASIC Compiler program, line 110 is not needed because the address of MYROUT will be assigned by the linking loader at load time.

2.3 CHAIN

Format: CHAIN [MERGE] <filename>[,[<line number exp>] [,ALL][,DELETE<range>]]

Purpose: To call a program and pass variables to it from the current program.

Remarks: <filename> is the name of the program that is called. Example:

```
CHAIN"PROG1"
```

<line number exp> is a line number or an expression that evaluates to a line number in the called program. It is the starting point for execution of the called program. If it is omitted, execution begins at the first line. Example:

```
CHAIN"PROG1",1000
```

<line number exp> is not affected by a RENUM command.

With the ALL option, every variable in the current program is passed to the called program. If the ALL option is omitted, the current program must contain a COMMON statement to list the variables that are passed. See Section 2.7. Example:

```
CHAIN"PROG1",1000,ALL
```

If the MERGE option is included, it allows a subroutine to be brought into the BASIC program as an overlay. That is, a MERGE operation is performed with the current program and the called program. The called program must be an ASCII file if it is to be MERGEd. Example:

```
CHAIN MERGE"OVRLAY",1000
```

After an overlay is brought in, it is usually desirable to delete it so that a new overlay may be brought in. To do this, use the DELETE option. Example:

```
CHAIN MERGE"OVRLAY2",1000,DELETE 1000-5000
```

The line numbers in <range> are affected by the RENUM command.

NOTE: The CHAIN statement with MERGE option leaves the files open and preserves the current OPTION BASE setting.

NOTE: If the MERGE option is omitted, CHAIN does not preserve variable types or user-defined functions for use by the chained program. That is, any DEFINT, DEFSNG, DEFDBL, DEFSTR, or DEFFN statements containing shared variables must be restated in the chained program.

NOTE: The Microsoft BASIC compiler does not support the ALL, MERGE, DELETE, and <line number exp> options to CHAIN. Thus, the statement format is CHAIN <filename>. If you wish to maintain compatibility with the Microsoft BASIC compiler, it is recommended that COMMON be used to pass variables and that overlays not be used. The CHAIN statement leaves the files open during CHAINing.

NOTE: When using the MERGE option, user-defined functions should be placed before any CHAIN MERGE statements in the program. Otherwise, the user-defined functions will be undefined after the merge is complete.

2.4 CLEAR

Format: CLEAR [,[<expression1>][,<expression2>]]

Purpose: To set all numeric variables to zero, all string variables to null, and to close all open files; and, optionally, to set the end of memory and the amount of stack space.

Remarks: <expression1> is a memory location which, if specified, sets the highest location available for use by BASIC.

<expression2> sets aside stack space for BASIC. The default is 256 bytes or one-eighth of the available memory, whichever is smaller.

NOTE: BASIC allocates string space dynamically. An "Out of string space error" occurs only if there is no free memory left for BASIC to use.

NOTE: The Microsoft BASIC Compiler supports the CLEAR statement with the restriction that <expression1> and <expression2> must be integer expressions. If a value of 0 is given for either expression, the appropriate default is used. The default stack size is 256 bytes, and the default top of memory is the current top of memory. The CLEAR statement performs the following actions:

Closes all files
Clears all COMMON and user variables
Resets the stack and string space
Releases all disk buffers

Examples:

```
CLEAR

CLEAR ,32768

CLEAR ,,2000

CLEAR ,32768,2000
```

2.5 CLOAD

Formats: CLOAD <filename>

CLOAD? <filename>

CLOAD* <array name>

Purpose: To load a program or an array from cassette tape into memory.

Remarks: CLOAD executes a NEW command before it loads the program from cassette tape. <filename> is the string expression or the first character of the string expression that was specified when the program was CSAVEd.

CLOAD? verifies tapes by comparing the program currently in memory with the file on tape that has the same filename. If they are the same, BASIC prints Ok. If not, BASIC prints NO GOOD.

CLOAD* loads a numeric array that has been saved on tape. The data on tape is loaded into the array called <array name> specified when the array was CSAVE*ed.

CLOAD and CLOAD? are always entered at command level as direct mode commands. CLOAD* may be entered at command level or used as a program statement. Make sure the array has been DIMensioned before it is loaded. BASIC always returns to command level after a CLOAD, CLOAD? or CLOAD* is executed. Before a CLOAD is executed, make sure the cassette recorder is properly connected and in the Play mode, and the tape is possitioned correctly.

See also CSAVE, Section 2.9.

NOTE: CLOAD and CSAVE are not included in all implementations of BASIC.

Example: CLOAD "MAX2"

Loads file "M" into memory.

2.6 CLOSE

Format: CLOSE[[#]<file number>[,[#]<file number...>]]

Purpose: To conclude I/O to a disk file.

Remarks: <file number> is the number under which the file was OPENed. A CLOSE with no arguments closes all open files.

The association between a particular file and file number terminates upon execution of a CLOSE. The file may then be reOPENed using the same or a different file number; likewise, that file number may now be reused to OPEN any file.

A CLOSE for a sequential output file writes the final buffer of output.

The END statement and the NEW command always CLOSE all disk files automatically. (STOP does not close disk files.)

Example: See Microsoft BASIC Disk I/O, in the Microsoft BASIC User's Guide.

2.7 COMMON

Format: COMMON <list of variables>

Purpose: To pass variables to a CHAINed program.

Remarks: The COMMON statement is used in conjunction with the CHAIN statement. COMMON statements may appear anywhere in a program, though it is recommended that they appear at the beginning. The same variable cannot appear in more than one COMMON statement. Array variables are specified by appending "()" to the variable name. If all variables are to be passed, use CHAIN with the ALL option and omit the COMMON statement.

Example:
```
100 COMMON A,B,C,D(),G$
110 CHAIN "PROG3",10
     .
     .
     .
```

NOTE: The Microsoft BASIC Compiler supports a modified version of the COMMON statement. The COMMON statement must appear in a program before any executable statements. The current non-executable statements are:

```
COMMON
DEFDBL, DEFINT, DEFSNG, DEFSTR
DIM
OPTION BASE
REM
%INCLUDE
```

Arrays in COMMON must be declared in preceding DIM statements,

The standard form of the COMMON statement is referred to as blank COMMON. FORTRAN style named COMMON areas are also supported; however, the variables are not preserved across CHAINs. The syntax for named COMMON is as follows:

```
COMMON /<name>/ <list of variables>
```

where <name> is 1 to 6 alphanumeric characters starting with a letter. This is useful for communicating with FORTRAN and assembly language routines without having to explicityly pass parameters in the CALL statement.

The blank COMMON size and order of variables must be the same in the CHAINing and CHAINed-to

programs. With the Microsoft BASIC Compiler, the best way to insure this is to place all blank COMMON declarations in a single include file and use the %INCLUDE statement in each program. For example:

```
MENU.BAS
        10 %INCLUDE COMDEF
        .
        .
        . 1000 CHAIN "PROG1"


PROG1.BAS
        10 %INCLUDE COMDEF
        .
        .
        . 2000 CHAIN "MENU"


COMDEF.BAS
        100 DIM A(100),B$(200)
        110 COMMON I,J,K,A,()
        120 COMMON A$,B$,(),X,Y,Z
```

2.8 CONT

Format: CONT

Purpose: To continue program execution after a Control-C has been typed, or a STOP or END statement has been executed.

Remarks: Execution resumes at the point where the break occurred. If the break occurred after a prompt from an INPUT statement, execution continues with the reprinting of the prompt (? or prompt string).

CONT is usually used in conjunction with STOP for debugging. When execution is stopped, intermediate values may be examined and changed using direct mode statements. Execution may be resumed with CONT or a direct mode GOTO, which resumes execution at a specified line number. CONT may be used to continue execution after an error.

CONT is invalid if the program has been edited during the break.

Example: See example Section 2.61, STOP.

2.9 CSAVE

Formats: CSAVE <string expression>

CSAVE* <array variable name>

Purpose: To save the program or an array currently in memory on cassette tape.

Remarks: Each program or array saved on tape is identified by a filename. When the command CSAVE <string expression> is executed, BASIC saves the program currently in memory on tape and uses the first character in <string expression> as the filename. <string expression> may be more than one character, but only the first character is used for the filename.

When the command CSAVE* <array variable name> is executed, BASIC saves the specified array on tape. The array must be a numeric array. The elements of a multidimensional array are saved with the leftmost subscript changing fastest.

CSAVE may be used as a program statement or as a direct mode command.

Before a CSAVE or CSAVE* is executed, make sure the cassette recorder is properly connected and in the Record mode.

See also CLOAD, Section 2.5.

NOTE: CSAVE and CLOAD are not included in all implementations of BASIC.

Example: CSAVE "TIMER"

Saves the program currently in memory on cassette under filename "T".

2.10 DATA

Format: DATA <list of constants>

Purpose: To store the numeric and string constants that are accessed by the program's READ statement(s). (See READ, Section 2.54)

Remarks: DATA statements are nonexecutable and may be placed anywhere in the program. A DATA statement may contain as many constants as will fit on a line (separated by commas), and any number of DATA statements may be used in a program. The READ statements access the DATA statements in order (by line number) and the data contained therein may be thought of as one continuous list of items, regardless of how many items are on a line or where the lines are placed in the program.

<list of constants> may contain numeric constants in any format, i.e., fixed point, floating point or integer. (No numeric expressions are allowed in the list.) String constants in DATA statements must be surrounded by double quotation marks only if they contain commas, colons or significant leading or trailing spaces. Otherwise, quotation marks are not needed.

The variable type (numeric or string) given in the READ statement must agree with the corresponding constant in the DATA statement.

DATA statements may be reread from the beginning by use of the RESTORE statement (Section 2.57).

Example: See examples in Section 2.54, READ.

2.11 DEF FN

Format: DEF FN<name>[(<parameter list>)]=<function definition>

Purpose: To define and name a function that is written by the user.

Remarks: <name> must be a legal variable name. This name, preceded by FN, becomes the name of the function. <parameter list> is comprised of those variable names in the function definition that are to be replaced when the function is called. The items in the list are separated by commas. <function definition> is an expression that performs the operation of the function. It is limited to one line. Variable names that appear in this expression serve only to define the function; they do not affect program variables that have the same name. A variable name used in a function definition may or may not appear in the parameter list. If it does, the value of the parameter is supplied when the function is called. Otherwise, the current value of the variable is used.

The variables in the parameter list represent, on a one-to-one basis, the argument variables or values that will be given in the function call.

User-defined functions may be numeric or string. If a type is specified in the function name, the value of the expression is forced to that type before it is returned to the calling statement. If a type is specified in the function name and the argument type does not match, a "Type mismatch" error occurs.

A DEF FN statement must be executed before the function it defines may be called. If a function is called before it has been defined, an "Undefined user function" error occurs. DEF FN is illegal in the direct mode.

Example:

```
      .
      .
410 DEF FNAB(X,Y)=X^3/Y^2
420 T=FNAB(I,J)
      .
      .
```

Line 410 defines the function FNAB. The function is called in line 420.

2.12 DEFINT/SNG/DBL/STR

Format: DEF<type> <range(s) of letters>
where <type> is INT, SNG, DBL, or STR

Purpose: To declare variable types as integer, single precision, double precision, or string.

Remarks: A DEFtype statement declares that the variable names beginning with the letter(s) specified will be that type variable. However, a type declaration character always takes precedence over a DEFtype statement in the typing of a variable.

If no type declaration statements are encountered, BASIC assumes all variables without declaration characters are single precision variables.

Examples:

```
10 DEFDBL L-P     All variables beginning with
                  the letters L, M, N, O, and P
                  will be double precision
                  variables.

10 DEFSTR A       All variables beginning with
                  the letter A will be string
                  variables.

10 DEFINT I-N,W-Z
                  All variable beginning with
                  the letters I, J, K, L, M,
                  N, W, X, Y, Z will be integer
                  variables.
```

2.13 DEF USR

Format: DEF USR[<digit>]=<integer expression>

Purpose: To specify the starting address of an assembly language subroutine.

Remarks: <digit> may be any digit from 0 to 9. The digit corresponds to the number of the USR routine whose address is being specified. If <digit> is omitted, DEF USR0 is assumed. The value of <integer expression> is the starting address of the USR routine. See Assembly Language Subroutines, in the Microsoft BASIC User's Guide.

Any number of DEF USR statements may appear in a program to redefine subroutine starting addresses, thus allowing access to as many subroutines as necessary.

Example:

```
 .
 .
 .
200 DEF USR0=24000
210 X=USR0(Y^2/2.89)
 .
 .
 .
```

2.14 DELETE

Format: DELETE[<line number>][-<line number>]

Purpose: To delete program lines.

Remarks: BASIC always returns to command level after a DELETE is executed. If <line number> does not exist, an "Illegal function call" error occurs.

Examples:

DELETE 40	Deletes line 40
DELETE 40-100	Deletes lines 40 through 100, inclusive
DELETE-40	Deletes all lines up to and including line 40

2.15 DIM

Format: DIM <list of subscripted variables>

Purpose: To specify the maximum values for array variable subscripts and allocate storage accordingly.

Remarks: If an array variable name is used without a DIM statement, the maximum value of its subscript(s) is assumed to be 10. If a subscript is used that is greater than the maximum specified, a "Subscript out of range" error occurs. The minimum value for a subscript is always 0, unless otherwise specified with the OPTION BASE statement (see Section 2.46).

The DIM statement sets all the elements of the specified arrays to an initial value of zero.

Example:
```
10 DIM A(20)
20 FOR I=0 TO 20
30 READ A(I)
40 NEXT I
    .
    .
    .
```

2.16 EDIT

Format: EDIT <line number>

Purpose: To enter Edit Mode at the specified line.

Remarks: In Edit Mode, it is possible to edit portions of a line without retyping the entire line. Upon entering Edit Mode, BASIC types the line number of the line to be edited, then it types a space and waits for an Edit Mode subcommand.

Edit Mode Subcommands

Edit Mode subcommands are used to move the cursor or to insert, delete, replace, or search for text within a line. The subcommands are not echoed. Most of the Edit Mode subcommands may be preceded by an integer which causes the command to be executed that number of times. When a preceding integer is not specified, it is assumed to be 1.

Edit Mode subcommands may be categorized according to the following functions:

1. Moving the cursor
2. Inserting text
3. Deleting text
4. Finding text
5. Replacing text
6. Ending and restarting Edit Mode

NOTE

In the descriptions that follow, <ch> represents any character, <text> represents a string of characters of arbitrary length, [i] represents an optional integer (the default is 1), and $ represents the Escape (or Altmode) key.

1. Moving the Cursor

   Space — Use the space bar to move the cursor to the right. [i]Space moves the cursor i spaces to the right. Characters are printed as you space over them.

   Rubout — In Edit Mode, [i]Rubout moves the cursor i spaces to the left (backspaces). Characters are printed as you backspace over them.

2. Inserting Text

   I — I<text>$ inserts <text> at the current cursor position. The inserted characters are printed on the terminal. To terminate insertion, type Escape. If Carriage Return is typed during an Insert command, the effect is the same as typing Escape and then Carriage Return. During an Insert command, the Rubout, Delete, or Underscore key on the terminal may be used to delete characters to the left of the cursor. Rubout will print out the characters as you backspace over them. Delete and Underscore will print an Underscore for each character that you backspace over. If an attempt is made to insert a character that will make the line longer than 255 characters, a bell (Control-G) is typed and the character is not printed.

   X — The X subcommand is used to extend the line. X moves the cursor to the end of the line, goes into insert mode, and allows insertion of text as if an Insert command had been given. When you are finished extending the line, type Escape or Carriage Return.

3. Deleting Text

   D — [i]D deletes i characters to the right of the cursor. The deleted characters are echoed between backslashes, and the cursor is positioned to the right of the last character deleted. If there are fewer than i characters to the right of the cursor, iD deletes the remainder of the line.

   H — H deletes all characters to the right of the cursor and then automatically enters insert mode. H is useful for replacing statements at the end of a line.

4. Finding Text

   S — The subcommand [i]S<ch> searches for the ith

occurrence of <ch> and positions the cursor before it. The character at the current cursor position is not included in the search. If <ch> is not found, the cursor will stop at the end of the line. All characters passed over during the search are printed.

K The subcommand [i]K<ch> is similar to [i]S<ch>, except all the characters passed over in the search are deleted. The cursor is positioned before <ch>, and the deleted characters are enclosed in backslashes.

5. Replacing Text

C The subcommand C<ch> changes the next character to <ch>. If you wish to change the next i characters, use the subcommand iC, followed by i characters. After the ith new character is typed, change mode is exited and you will return to Edit Mode.

6. Ending and Restarting Edit Mode

<cr> Typing Carriage Return prints the remainder of the line, saves the changes you made and exits Edit Mode.

E The E subcommand has the same effect as Carriage Return, except the remainder of the line is not printed.

Q The Q subcommand returns to BASIC command level, without saving any of the changes that were made to the line during Edit Mode.

L The L subcommand lists the remainder of the line (saving any changes made so far) and repositions the cursor at the beginning of the line, still in Edit Mode. L is usually used to list the line when you first enter Edit Mode.

A The A subcommand lets you begin editing a line over again. It restores the original line and repositions the cursor at the beginning.

NOTE

If BASIC receives an unrecognizable command or illegal character while in Edit Mode, it prints a bell (Control-G) and the command or character is ignored.

Syntax Errors

When a Syntax Error is encountered during execution of a program, BASIC automatically enters Edit Mode at the line that caused the error. For example:

```
10 K = 2(4)
RUN
?Syntax error in 10
10
```

When you finish editing the line and type Carriage Return (or the E subcommand), BASIC reinserts the line, which causes all variable values to be lost. To preserve the variable values for examination , first exit Edit Mode with the Q subcommand. BASIC will return to command level, and all variable values will be preserved.

Control-A

To enter Edit Mode on the line you are currently typing, type Control-A. BASIC responds with a carriage return, an exclamation point (!) and a space. The cursor will be positioned at the first character in the line. Proceed by typing an Edit Mode subcommand.

NOTE

Remember, if you have just entered a line and wish to go back and edit it, the command "EDIT." will enter Edit Mode at the current line. (The line number symbol "." always refers to the current line.)

2.17 END

Format: END

Purpose: To terminate program execution, close all files and return to command level.

Remarks: END statements may be placed anywhere in the program to terminate execution. Unlike the STOP statement, END does not cause a BREAK message to be printed. An END statement at the end of a program is optional. BASIC always returns to command level after an END is executed.

Example:
```
520 IF K>1000 THEN END ELSE GOTO 20
```

2.18 ERASE

Format: ERASE <list of array variables>

Purpose: To eliminate arrays from a program.

Remarks: Arrays may be redimensioned after they are ERASEd, or the previously allocated array space in memory may be used for other purposes. If an attempt is made to redimension an array without first ERASEing it, a "Redimensioned array" error occurs.

NOTE: The Microsoft BASIC compiler does not support ERASE.

Example:

```
.
.
.
450 ERASE A,B
460 DIM B(99)
.
.
.
```

2.19 ERR AND ERL VARIABLES

When an error handling subroutine is entered, the variable ERR contains the error code for the error, and the variable ERL contains the line number of the line in which the error was detected. The ERR and ERL variables are usually used in IF...THEN statements to direct program flow in the error trap routine.

If the statement that caused the error was a direct mode statement, ERL will contain 65535. To test if an error occurred in a direct statement, use IF 65535 = ERL THEN ...
Otherwise, use

```
IF ERR = error code THEN ...

IF ERL = line number THEN ...
```

If the line number is not on the right side of the relational operator, it cannot be renumbered by RENUM. Because ERL and ERR are reserved variables, neither may appear to the left of the equal sign in a LET (assignment) statement. The Microsoft BASIC error codes are listed in Appendix A.

2.20 ERROR

Format: ERROR <integer expression>

Purpose: 1) To simulate the occurrence of a BASIC error; or 2) to allow error codes to be defined by the user.

Remarks: The value of <integer expression> must be greater than 0 and less than 255. If the value of <integer expression> equals an error code already in use by BASIC (see Appendix A), the ERROR statement will simulate the occurrence of that error, and the corresponding error message will be printed. (See Example 1.)

To define your own error code, use a value that is greater than any used by the Microsoft BASIC error codes. (It is preferable to use the highest available values, so compatibility may be maintained when more error codes are added to Microsoft BASIC.) This user-defined error code may then be conveniently handled in an error trap routine. (See Example 2.)

If an ERROR statement specifies a code for which no error message has been defined, BASIC responds with the message UNPRINTABLE ERROR. Execution of an ERROR statement for which there is no error trap routine causes an error message to be printed and execution to halt.

Example 1:
```
LIST
10 S = 10
20 T = 5
30 ERROR S + T
40 END
Ok
RUN
String too long in line 30
```

Or, in direct mode:

```
Ok
ERROR 15              (you type this line)
String too long       (BASIC types this line)
Ok
```

Example 2:

```
 .
 .
 .
110 ON ERROR GOTO 400
120 INPUT "WHAT IS YOUR BET";B
130 IF B > 5000 THEN ERROR 210
 .
 .
 .
400 IF ERR = 210 THEN PRINT "HOUSE LIMIT IS $5000"
410 IF ERL = 130 THEN RESUME 120
 .
 .
 .
```

2.21 FIELD

Format: FIELD[#]<file number>,<field width> AS <string variable>...

Purpose: To allocate space for variables in a random file buffer.

Remarks: To get data out of a random buffer after a GET or to enter data before a PUT, a FIELD statement must have been executed.

<file number> is the number under which the file was OPENed. <field width> is the number of characters to be allocated to <string variable>. For example,

FIELD 1, 20 AS N$, 10 AS ID$, 40 AS ADD$

allocates the first 20 positions (bytes) in the random file buffer to the string variable N$, the next 10 positions to ID$, and the next 40 positions to ADD$. FIELD does NOT place any data in the random file buffer. (See LSET/RSET and GET.)

The total number of bytes allocated in a FIELD statement must not exceed the record length that was specified when the file was OPENed. Otherwise, a "Field overflow" error occurs. (The default record length is 128.)

Any number of FIELD statements may be executed for the same file, and all FIELD statements that have been executed are in effect at the same time.

Example: See Microsoft BASIC Disk I/O, in the Microsoft BASIC User's Guide.

NOTE: Do not use a FIELDed variable name in an INPUT or LET statement. Once a variable name is FIELDed, it points to the correct place in the random file buffer. If a subsequent INPUT or LET statement with that variable name is executed, the variable's pointer is moved to string space.

2.22 FOR...NEXT

Format:

```
FOR <variable>=x TO y [STEP z]
 .
 .
 .
NEXT [<variable>][,<variable>...]
```

where x, y and z are numeric expressions.

Purpose: To allow a series of instructions to be performed in a loop a given number of times.

Remarks: <variable> is used as a counter. The first numeric expression (x) is the initial value of the counter. The second numeric expression (y) is the final value of the counter. The program lines following the FOR statement are executed until the NEXT statement is encountered. Then the counter is incremented by the amount specified by STEP. A check is performed to see if the value of the counter is now greater than the final value (y). If it is not greater, BASIC branches back to the statement after the FOR statement and the process is repeated. If it is greater, execution continues with the statement following the NEXT statement. This is a FOR...NEXT loop. If STEP is not specified, the increment is assumed to be one. If STEP is negative, the final value of the counter is set to be less than the initial value. The counter is decremented each time through the loop, and the loop is executed until the counter is less than the final value.

The body of the loop is skipped if the initial value of the loop times the sign of the step exceeds the final value times the sign of the step.

Nested Loops

FOR...NEXT loops may be nested, that is, a FOR...NEXT loop may be placed within the context of another FOR...NEXT loop. When loops are nested, each loop must have a unique variable name as its counter. The NEXT statement for the inside loop must appear before that for the outside loop. If nested loops have the same end point, a single NEXT statement may be used for all of them.

The variable(s) in the NEXT statement may be omitted, in which case the NEXT statement will match the most recent FOR statement. If a NEXT